Rust basics

PHP to Rust — full chapter (all sections)

Variables

In PHP

A variable is mutable, and its name lives for the whole function. An `if` block does not hide it. After the brace, `$points` is still `2`.

`strlen("five")` is `4`.

`const` is a compile-time name. On PHP 8.3, assigning to it is a parse error: `syntax error, unexpected token "="`. A `readonly` property is different: the constructor sets it, and a later assignment throws `Cannot modify readonly property`.

score.php

```
 <?php

function score(): void
{
    $points = 1;
    if (true) {
        $points = 2;
    }
    echo $points, "\n";

    $count = "five";
    $count = strlen($count);
    echo $count, "\n";
}

score();

const MAX_COUNT = 100;
echo MAX_COUNT, "\n";
```

That prints `2`, then `4`, then `100`.

constants.php

```
<?php

const MAX_COUNT = 100;
MAX_COUNT = 200;
```

PHP 8.3 rejects that file before it runs: `Parse error: syntax error, unexpected token "="`.

Config.php

```
<?php

readonly class Config
{
    public function __construct(public int $maxCount)
    {
    }
}

$config = new Config(100);
echo $config->maxCount, "\n";
```

`$config->maxCount = 200` throws `Error: Cannot modify readonly property Config::$maxCount`.

In Rust

`let` binds a name. Without `mut`, a second assignment is a compile error. `let` again, in the same scope, creates a new binding. That is how a string becomes a number without a mutable variable: `parse` is the usual reason to shadow. The old binding is hidden for the rest of that scope.

A shadow inside a block ends at that block's `}`. The outer name is still there.

`const` is an item, not a `let` binding. It has a type, the name is `SCREAMING_CASE`, the value is computed at compile time, and it is never `mut`.

src/main.rs

```
const MAX_POINTS: u32 = 100_000;

fn main() {
    let x = 5;
    println!("{x}");
```

```
    let mut counter = 0;
    counter += 1;
    println!("{counter}");

    let guess = "42";
    let guess: u32 = guess.parse().expect("guess should be a number");
    println!("{guess}");

    let count = "five";
    let count = count.len();
    println!("{count}");

    let name = 5;
    {
        let name = name + 1;
        println!("inner {name}");
    }
    println!("outer {name}");

    println!("{MAX_POINTS}");
}
```

`count` prints `4`. `inner` prints `6`. `outer` prints `5`.

> **Watch for:** A shadow inside a block ends at that block's `}`. The outer name comes back. In PHP the assignment would still be visible, because the name belongs to the function.

> **Idiomatic Rust:** Shadow with `parse` when a name should change type and stay immutable. Use `const` for a typed value the compiler can compute up front.

Side by side

	PHP	Rust
Second assignment	Updates the same variable	Needs `let mut`, or a new `let`
`"five"` length	`strlen` is `4`	`.len()` is `4`

	PHP	Rust
Name inside `if` or `{ }`	Still visible after the block	A shadow lasts only until `}`
Constant	`const`. Assigning to it is a parse error	`const NAME: Type = value`. Never `mut`
Immutable field	`readonly` property	Fields are immutable unless the whole value is `let mut`

A name you bind but never read is an unused-variable warning. Prefix it with `_` (`let _scratch = 1;`) to silence that, the same idea as intentionally unused parameters. You can also declare without a value and assign once: `let x; x = 5;`. That is still one immutable binding, not `mut`.

Try it

Open `src/main.rs` in the Playground. Change `"42"` to `"four"` and run it. `expect` panics because the parse failed. Put `"42"` back. Under `let x = 5;`, add `x = 6;` and read the compiler error. It is `E0384` (cannot assign twice to immutable variable). Remove that line. `outer` still prints `5`.

Key takeaways

- `let` is immutable. `let mut` is the assignment that changes one binding. A second assignment without `mut` is `E0384`.
- Shadowing is a new `let` in the same scope. `parse` is the usual reason. A shadow inside `{ }` ends at `}`.
- `const` is a typed item in `SCREAMING_CASE`. It is never `mut`. PHP reports `syntax error, unexpected token "="` if you assign to a `const`.
- `_name` silences unused-variable warnings. `let x; x = 5;` is deferred init of one immutable binding.

What's next: Numbers, bool and char

Numbers, bool and char

In PHP

PHP has one `int` and one `float`. `7 / 2` is `3.5`. `intdiv(7, 2)` is `3`. `PHP_INT_MAX + 1` becomes a float. The type grows.

These values are falsy: `0`, `0.0`, `""`, `"0"`, `[]`, `null`, and `false`. There is no `char`. `"é"` is a string, and `strlen` counts bytes.

numbers.php

```
<?php

echo 7 / 2, "\n";
echo intdiv(7, 2), "\n";

$ready = "0";
if ($ready) {
    echo "truthy\n";
} else {
    echo "falsy\n";
}

echo strlen("é"), "\n";
```

That prints `3.5`, then `3`, then `falsy`, then `2`.

In Rust

An unsuffixed integer is `i32`. An unsuffixed float is `f64`. A suffix picks another width (`1_000_000u64`, `2.5_f32`). Underscores are only for reading.

`isize` and `usize` match the pointer width. A length or an index is a `usize`.

	8 bits	16	32	64	128
Signed	`i8`	`i16`	`i32`, the default	`i64`	`i128`
Unsigned	`u8`	`u16`	`u32`	`u64`	`u128`

A signed `iN` runs from `-2^(N-1)` through `2^(N-1) - 1`. An unsigned `uN` runs from `0` through `2^N - 1`. `i8` is `-128` through `127`. `u8` is `0` through `255`.

Integer `/` truncates. `7 / 2` is `3`.

In a debug build, an overflowing `+` panics. A release build wraps. `checked_add` returns `None` either way. Next to it, `wrapping_add` always wraps and `saturating_add` clamps at the type's min or max. The calculator uses `checked_add`, `checked_sub`, `checked_mul`, and `checked_div` so overflow and divide-by-zero both come back as `None`.

`as` between integers keeps the low bits. `300 as u8` is `44`. When the value might not fit, use `try_into` (or `TryFrom`). `as` from a float rounds toward zero, so `300.9 as i32` is `300`. A float that cannot fit saturates to the integer's min or max, and `NaN` becomes `0`. That cast does not report the loss.

Do not compare floats with `==`. `0.1 + 0.2` is not a clean `0.3`. Check an explicit tolerance instead: `(0.1 + 0.2 - 0.3).abs() < 1e-9`. `f64::EPSILON` is the gap between `1.0` and the next `f64`, so it is not a general-purpose tolerance.

`if` needs a `bool`. A number is not a condition.

`char` is four bytes and one Unicode scalar value. It is not a PHP string, and it is not a grapheme cluster.

src/main.rs

```
 use std::f64::consts::PI;

fn main() {
    let n = 42;
    let width = 1.5;
    let big = 1_000_000u64;
    let single = 2.5_f32;
    let index: usize = 0;
    println!("{n} {width} {big} {single} {index}");

    let quotient = 7 / 2;
    println!("{quotient}");

    let overflow = i32::MAX.checked_add(1);
    println!("{overflow:?}");
```

```
    let wide: i32 = 300;
    let truncated = wide as u8;
    println!("{truncated}");

    let fitted: Result<u8, _> = 200i32.try_into();
    let too_big: Result<u8, _> = wide.try_into();
    println!("{fitted:?} {too_big:?}");

    let toward_zero = 300.9_f64 as i32;
    println!("{toward_zero}");

    println!("{PI}");

    let ready = true;
    if ready {
        println!("ready");
    }

    let letter: char = 'é';
    let han: char = '?';
    let bytes = size_of::<char>();
    println!("{letter} {han} {bytes}");

    let close = (0.1_f64 + 0.2 - 0.3).abs() < 1e-9;
    println!("{close}");
}
```

`quotient` is `3`. `overflow` is `None`. `truncated` is `44`. `fitted` is `Ok(200)` and `too_big` is `Err`. `toward_zero` is `300`. `bytes` is `4`. `close` is `true`.

> **Watch for:** `as` keeps going when the value does not fit. `try_into` returns `Err` instead. A float compared with `==` will not do what the decimal looks like it should. `f64::EPSILON` is not a drop-in tolerance for every magnitude.

> **Idiomatic Rust:** `checked_add` when overflow matters, as in the calculator. `wrapping_add` and `saturating_add` when you want wrap or clamp instead of `None`. `usize` for an index. `std::f64::consts::PI` for pi. `(a - b).abs() < 1e-9` (or another tolerance you choose) for float equality checks.

Side by side

	PHP	Rust
`7 / 2`	`3.5`. `intdiv` is `3`	`3`

	PHP	Rust
Overflow	The int becomes a float	Debug panics, release wraps, `checked_add` is `None`
Narrowing	The value may change type	`as` truncates. `try_into` when it might not fit
Float equality	Often `abs($a - $b) < 1e-9`	`(a - b).abs() < 1e-9`. Not `f64::EPSILON` as a general tolerance
`"0"` or `[]` in `if`	Falsy	`if` needs a `bool`
One letter	A string. `strlen("é")` is `2`	`char`, four bytes, one scalar

Try it

Open `src/main.rs` in the Playground. Confirm `7 / 2` prints `3` and `checked_add` prints `None`. Change `wide` from `300` to `40` and run it again. `as` and `try_into` then agree on `40`. Put `300` back: `as` prints `44` and `try_into` is `Err`. Confirm `close` prints `true` for the `1e-9` tolerance check.

Key takeaways

- Unsuffixed integers are `i32` and unsuffixed floats are `f64`. Width is the number of bits. `usize` is the index type.
- `7 / 2` is `3`. Overflow is `checked_add` when you need a `None`. Prefer `wrapping_add` or `saturating_add` when wrap or clamp is the rule.
- `as` truncates. `try_into` fails when the integer does not fit. Compare floats with an explicit tolerance such as `1e-9`, not `==`. `f64::EPSILON` is not a general-purpose tolerance.
- `bool` has no truthiness. `char` is one Unicode scalar, four bytes wide.

What's next: Functions and expressions

Functions and expressions

In PHP

A function names its parameters and, when you want it, its return type. `return` sends the value back. A function declared `: void` still produces `null` if you use the call as a value. `return null;` inside that function is a compile error. Leave early with a bare `return;`.

area.php

```
 <?php

function area(int $width, int $height): int
{
    return $width * $height;
}

function greet(string $name): void
{
    echo "Hello, $name\n";
}

echo area(3, 4), "\n";
greet("Ada");

$result = greet("Ada");
var_dump($result);
```

`var_dump` prints `NULL`. The call's value is `null`.

Four PHP habits, and the Rust stand-in for each:

- Default parameter values. Rust uses `Option<T>`, or a second function.
- Named arguments. Rust uses a struct or a builder. Structs are chapter 4.
- Overloading. PHP does not allow two functions with one name either. Rust uses different names, or generics later.
- Variadics (`...$args`). Rust uses a slice or a `Vec`.

A function declared inside another PHP function is registered in the global namespace the first time the outer function runs. Rust's nested `fn` stays inside the outer function, and it cannot see that function's local variables. Closures, which can, come later.

In Rust

`fn` names are `snake_case`. Every parameter has a type. A function that returns a value says so with `->`. The last expression, with no semicolon, is the return value. A semicolon turns that line into a statement, the function's value becomes `()`, and the compiler reports `expected i32, found ()`.

`return` leaves early. A block is an expression too: `let y = { ... };` uses the block's last line as the value of `y`.

A nested `fn` is an item. It does not capture local variables. Pass in what it needs.

src/main.rs

```
 fn area(width: i32, height: i32) -> i32 {
    width * height
}

fn greet(name: &str) {
    println!("Hello, {name}");
}

fn double_or_zero(n: i32) -> i32 {
    if n < 0 {
        return 0;
    }
    n * 2
}

fn main() {
    let value = area(3, 4);
    println!("{value}");

    greet("Ada");

    let y = {
        let part = 2;
        part + 3
    };
    println!("{y}");

    println!("{}", double_or_zero(5));
    println!("{}", double_or_zero(-1));

    let count = 2;
    fn announce(count: i32) {
        println!("count is {count}");
    }
    announce(count);
```

```
}
```

`area` prints `12`. `y` prints `5`. `double_or_zero(-1)` prints `0`. `greet` returns `()`.

Naming the outer local inside the nested function without passing it is `E0434`. This program must fail:

src/main.rs · must fail

```
fn main() {
    let count = 2;
    fn announce() {
        println!("count is {count}");
    }
    announce();
}
```

```
error[E0434]: can't capture dynamic environment in a fn item
 --> src/main.rs:4:28
  |
4 |         println!("count is {count}");
  |                            ^^^^^^^
  |
  = help: use the `|| { ... }` closure form instead
```

Add a semicolon and `area` no longer returns `i32`. This is the compiler's text:

```
error[E0308]: mismatched types
 --> src/main.rs:1:37
  |
1 | fn area(width: i32, height: i32) -> i32 {
  |    ----                             ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
2 |     width * height;
  |                   - help: remove this semicolon to return this value
```

> **Watch for:** The semicolon. `width * height` is an `i32`. `width * height;` is `()`. The error points at the `-> i32` and says `expected i32, found ()`.

> **Idiomatic Rust:** Leave the tail expression without a semicolon. Use `return` only when you leave before the last line. Pass a nested `fn` what it needs. It cannot see the caller's locals.

Side by side

	PHP	Rust
Send a value back	`return`	The last expression. `return` for an early exit
A trailing semicolon	Still needs `return`	The value becomes `()`
No useful value	`: void`, and the call yields `null`	The type is `()`
Default, named, variadic	Built into the call	`Option` or another function, a struct, a slice or `Vec`
Function inside a function	Often becomes global	Nested `fn` cannot capture locals

Try it

Open the first `src/main.rs` in the Playground. Add a semicolon after `width * height` and read the error. It says `expected i32, found ()`. Remove the semicolon. `area(3, 4)` prints `12` again.

Key takeaways

- Parameters are typed, names are `snake_case`, and `->` is the return type.
- The last expression without `;` is the return value. With `;` the type is `()`, and the error is `expected i32, found ()`.
- `return` leaves early. `{ ... }` is an expression. `()` is a value. A PHP `: void` call yields `null`.
- A nested `fn` cannot capture local variables. Closures come later.

What's next: Control flow

Control flow

In PHP

`if` checks truthiness. These values are falsy: `0`, `0.0`, `""`, `"0"`, `[]`, `null`, and `false`. The same list is in the numbers lesson. A ternary is the expression form of `if`.

`range(5, 1)` counts down: `5 4 3 2 1`. A C-style `for` keeps the index in `$i`. `break 2` leaves two loops. `do { } while` runs the body, then tests.

flow.php

```
 <?php

$n = 7;
$label = $n > 0 ? "positive" : "non-positive";
echo $label, "\n";

if ("0") {
    echo "truthy\n";
} else {
    echo "falsy\n";
}

foreach (range(5, 1) as $value) {
    echo $value, " ";
}
echo "\n";

$nums = [10, 20, 30];
for ($i = 0; $i < count($nums); $i++) {
    echo $i, ": ", $nums[$i], "\n";
}

for ($x = 0; $x < 3; $x++) {
    for ($y = 0; $y < 3; $y++) {
        if ($x === 1 && $y === 1) {
            break 2;
        }
        echo "$x,$y\n";
    }
}
```

In Rust

`if` is an expression. Both branches have the same type, and the condition is a `bool`. `if 1` does not compile. `if "0"` does not compile.

`loop` runs until `break`. `break` can carry the value of the loop. `while` tests a `bool` before each pass. `for` walks a range or a shared collection (`for item in &nums`).

`1..4` is `1, 2, 3`. `1..=5` includes `5`. `(1..=5).rev()` counts down. `5..1` is an empty range: **rustc accepts it**, and the loop body never runs. Clippy is what rejects it, via `clippy::reversed_empty_ranges`, which is deny by default when you run `cargo clippy` (or a setup that enables Clippy):

```
 error: this range is empty so it will yield no values
  --> src/main.rs:2:18
   |
2 |     for value in 5..1 {
   |                  ^^^^
   |
   = note: `#[deny(clippy::reversed_empty_ranges)]` on by default
help: consider using the following if you are attempting to iterate over this
range in reverse
   |
2 -     for value in 5..1 {
2 +     for value in (1..5).rev() {
```

`(1..5).rev()` is `4, 3, 2, 1`. Include both ends, as PHP `range(5, 1)` does, with `(1..=5).rev()`.

`.enumerate()` gives the index. A label such as `'outer:` lets `break 'outer` leave the outer loop. PHP's `break 2` is that idea. `continue` skips the rest of this pass. `continue 'outer` starts the next pass of the labelled loop. Rust has no `do while`. A `loop` with `break` at the end of the body is that shape.

`while let` is the idiomatic loop for "keep going while this pattern matches". PHP developers often miss it. Popping a stack until it is empty is:

```
 let mut stack = vec![1, 2, 3];
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

src/main.rs

```
fn main() {
    let n = 7;
    let label = if n > 0 { "positive" } else { "non-positive" };
    println!("{label}");

    let mut ticks = 0;
    let found = loop {
        ticks += 1;
        if ticks == 4 {
            break ticks;
        }
    };
    println!("{found}");

    let mut i = 0;
    while i < 3 {
        i += 1;
    }
    println!("{i}");

    for value in 1..4 {
        println!("{value}");
    }
    for value in (1..=5).rev() {
        println!("rev {value}");
    }

    let nums = [10, 20, 30];
    for item in &nums {
        println!("{item}");
    }
    for (index, item) in nums.iter().enumerate() {
        println!("{index}: {item}");
    }

    'outer: for x in 0..3 {
        for y in 0..3 {
            if x == 1 && y == 1 {
                break 'outer;
            }
            println!("{x},{y}");
        }
    }

    let mut step = 0;
    loop {
        step += 1;
        println!("step {step}");
        if step == 2 {
            break;
        }
    }

    let mut stack = vec![1, 2, 3];
```

```
    while let Some(top) = stack.pop() {
        println!("pop {top}");
    }
}
```

`label` is `positive`. `found` is `4`. `1..4` prints `1`, `2`, `3`. The countdown prints `rev 5` through `rev 1`. The labelled loops print `0,0`, `0,1`, `0,2`, and `1,0`, then stop. `step` prints `1` and `2`. The stack prints `pop 3`, `pop 2`, `pop 1`.

> **Watch for:** `5..1` compiles with `rustc` and yields nothing. Clippy denies it. Count down with `(1..=5).rev()`. An `if` condition is a `bool`, including the falsy PHP values `"0"` and `[]`.

> **Idiomatic Rust:** `for item in &nums`, and `.enumerate()` when you need the index. `while let` for pattern loops. `break 'outer` / `continue 'outer` on a labelled loop. `break` with a value when the loop's result is a number.

Side by side

	PHP	Rust
`if` as a value	Ternary	`if` / `else`, both branches the same type
Condition	Falsy list, including `"0"` and `[]`	A `bool`
Count down	`range(5, 1)`	`(1..=5).rev()`. `5..1` is empty (clippy denies it)
Index	`for ($i = 0; ...)`	`.enumerate()`
Leave two loops	`break 2`	`break 'outer`
Pattern loop	`while (($x = array_pop(...)) !== null)`	`while let Some(x) = ...`

	PHP	Rust
`do while`	Body, then the test	`loop`, with `break` at the end

Try it

Open `src/main.rs` in the Playground. `1..4` prints `1`, `2`, and `3`. `(1..=5).rev()` prints `5` down to `1`. Change `break 'outer` to `break` and run it again. The inner loop stops at `1,1`, and the outer loop continues with `2,0`.

Key takeaways

- `if` is an expression. The condition is a `bool`, and both branches have one type.
- `loop` can `break` a value. `while` tests first. `while let` keeps going while the pattern matches. `for` walks `..`, `..=`, or `&collection`.
- `5..1` compiles with `rustc` and yields nothing. Clippy denies `reversed_empty_ranges`. `(1..=5).rev()` counts down the way `range(5, 1)` does. `.enumerate()` replaces `$i`.
- `'outer:` with `break 'outer` or `continue 'outer` targets the labelled loop. There is no `do while`. Put `break` at the end of a `loop`.

What's next: Tuples and arrays

Tuples and arrays

In PHP

A list can mix types. `[$left, $right] = pair()` takes the two values a function returns. `count` is the length, and the length can change later by appending. Nothing in the type says "this list has 5 elements".

pairs.php

```
 <?php

function pair(int $left, int $right): array
{
    return [$left + $right, $left * $right];
}

[$sum, $product] = pair(3, 4);
echo "$sum $product\n";

$point = [10, 2.5, "origin"];
echo $point[0], " ", $point[1], " ", $point[2], "\n";

$zeros = array_fill(0, 5, 0);
echo count($zeros), "\n";
```

In Rust

A tuple packs mixed types. `.0`, `.1`, and `.2` read the fields. Destructuring names them. A function returns two values as one tuple, the way `[$sum, $product] = pair()` does in PHP.

An array `[T; N]` is one type of element, and the length is part of the type. `[0; 5]` is five zeros. `.len()` is the length. `for n in &numbers` walks the elements.

`numbers[10]` does not compile. `let index = 10; numbers[index]` does not compile either: rustc can see the `10`.

```
 error: this operation will panic at runtime
  --> src/main.rs:3:20
   |
3  |     println!("{}", numbers[10]);
   |                    ^^^^^^^^^^^ index out of bounds: the length is 5 but
```

```
the index is 10
  |
  = note: `#[deny(unconditional_panic)]` on by default
```

An index the compiler cannot fold still compiles, then panics when it runs. `std::hint::black_box(10)` is one way to hide a value from rustc's const analysis:

```
 thread 'main' panicked at src/main.rs:4:20:
index out of bounds: the len is 5 but the index is 10
```

`.get(10)` returns `Option`. Here it is `None`.

A slice is a view of contiguous elements: `&[T]`. `&numbers[1..3]` is a two-element borrow of the array (indexes 1 and 2). The next lessons pass `&[Option<i32>]` the same way. Chapter 3 goes deeper on borrowing. When the length has to change, use `Vec`. That is the next lesson.

src/main.rs

```
 fn pair(left: i32, right: i32) -> (i32, i32) {
    (left + right, left * right)
}

fn main() {
    let point = (10, 2.5, "origin");
    let x0 = point.0;
    let y0 = point.1;
    let label0 = point.2;
    println!("{x0} {y0} {label0}");

    let (x, y, label) = point;
    println!("{x} {y} {label}");

    let (sum, product) = pair(3, 4);
    println!("{sum} {product}");

    let zeros = [0; 5];
    let len = zeros.len();
    println!("{len} {zeros:?}");

    let numbers = [10, 20, 30, 40, 50];
    for n in &numbers {
        println!("{n}");
    }

    let index = 1;
    let chosen = numbers[index];
    println!("{chosen}");
```

```
    let missing = numbers.get(10);
    println!("{missing:?}");

    let mid: &[i32] = &numbers[1..3];
    println!("{mid:?}");
}
```

`pair(3, 4)` prints `7` and `12`. `zeros` prints `5 [0, 0, 0, 0, 0]`. `chosen` is `20`. `missing` is `None`. `mid` is `[20, 30]`.

To see the runtime panic, replace `let index = 1;` with `let index = std::hint::black_box(10);` and run it. The index is 10, the length is 5, and the program panics.

> **Watch for:** `numbers[10]` is a compile error on a `[T; N]`. A `let` that rustc can see through is the same error. Hide the value with `black_box` (or parse it from args) when you need a real runtime panic. `.get` is the call that returns `None` instead of panicking.

> **Idiomatic Rust:** Return `(sum, product)` when a function has two results. `for n in &numbers` to walk an array. `&numbers[1..3]` for a slice. `.get(index)` when the index comes from outside the program.

Side by side

	PHP	Rust
Mixed values	One array	A tuple, read with `.0` or destructuring
Two return values	`return [a, b]` and `[$a, $b] = f()`	`-> (i32, i32)`
Fixed list	`array_fill`, length can still change	`[T; N]`. `[0; 5]`

	PHP	Rust
Out of range	A warning, then `null`	Constant index: compile error. Opaque runtime index: panic. `.get` is `Option`
Slice of a list	`array_slice`	`&numbers[1..3]` is `&[T]`
Growing list	`$a[] = 1`	`Vec` , next lesson

Try it

Open `src/main.rs` in the Playground. `chosen` prints `20` and `missing` prints `None`. Change `index` to `std::hint::black_box(10)` and run it. The program panics: the length is 5 and the index is 10. Put `1` back.

Key takeaways

- A tuple holds mixed types. `.0` reads a field, and destructuring names the fields. Two return values are one tuple.
- `[T; N]` fixes the length in the type. `[0; 5]` repeats a value. `for n in &numbers` walks it. `&numbers[1..3]` is a slice (`&[T]`).
- `numbers[10]` does not compile. An index rustc cannot precompute panics. `.get` returns `Option` .
- A list whose length changes is a `Vec` .

What's next: Vec and HashMap

Vec and HashMap

In PHP

One array does both jobs. `$numbers[] = 10` grows a list. `$scores["Ada"] = 10` stores a value under a string key. Keys keep insertion order. A missing key is handled with `??`.

counts.php

```
 <?php

$numbers = [];
$numbers[] = 10;
$numbers[] = 30;
$numbers[] = 20;
echo $numbers[0], "\n";

$counts = [];
foreach (preg_split("/\s+/", "the cat and the dog") as $word) {
    $counts[$word] = ($counts[$word] ?? 0) + 1;
}
foreach ($counts as $word => $count) {
    echo "$word $count\n";
}
```

The counts print in insertion order: `the`, `cat`, `and`, `dog`.

`array_map` and `array_filter` become iterator chains. Iterators have their own chapter. A nested array used as a record becomes a struct (chapter 4). JSON arrives through serde, in the API chapter.

In Rust

A list is a `Vec<T>`. `vec![]` starts it. `push`, `pop`, `len`, `is_empty`, and `contains` are the usual operations. Every element has the same type. Mixed types need an enum (chapter 4).

`numbers[0]` reads an element. On a `Vec`, `numbers[9]` compiles and then panics, because the length is not part of the type:

```
 thread 'main' panicked at src/main.rs:3:27:
index out of bounds: the len is 3 but the index is 9
```

`numbers.get(9)` is `None`. `for n in &numbers` reads. `for n in &mut numbers` writes. `sort` orders the values. `sort_by_key` orders by one field.

A string-keyed array is a `HashMap<String, V>`. Build the key with `.to_string()`. `insert` stores it. `get("Ada")` looks it up and returns `Option`. `contains_key` and `remove` do what they say. Walking the map visits every pair.

`*n += 1` and `*counts.entry(...)` : `*` writes through to the value the loop or entry points at. The Rust side of `$counts[$word] = ($counts[$word] ?? 0) + 1` is `*counts.entry(word.to_string()).or_insert(0) += 1`. `entry(...).or_default()` is the same idea when the value type has a useful `Default` (for example `0` for integers). `retain` keeps only matching entries. `extend` appends from another iterator. The `String` is how the map owns the word. `get("Ada")` looks up with the text of the key.

A set of unique values is a `HashSet`. PHP often fakes that with `array_unique` or `array_flip` so the keys act as the set. Rust has a real set type: `use std::collections::HashSet;` then `HashSet::from([1000, 2500, 800])`.

> **Watch for:** A PHP array keeps insertion order. `HashMap` iteration order is unspecified, and it can change between runs. `BTreeMap` walks keys in sorted order. The `indexmap` crate keeps insertion order when that is the behaviour you need.

`contains(&30)` : `&` lends the value for the call. Chapter 3 explains. `|row| row.1` is a closure, a small inline function. Closures get their own lesson.

src/main.rs

```
 use std::collections::{BTreeMap, HashMap};

fn main() {
    let mut numbers: Vec<i32> = vec![];
    let empty = numbers.is_empty();
    println!("{empty}");
    numbers.push(10);
    numbers.push(30);
    numbers.push(20);
    let len = numbers.len();
    let has_30 = numbers.contains(&30);
    let first = numbers[0];
    println!("{len} {has_30} {first}");
```

```
    let missing = numbers.get(9);
    println!("{missing:?}");

    for n in &numbers {
        println!("item {n}");
    }
    for n in &mut numbers {
        *n += 1;
    }
    numbers.sort();
    println!("{numbers:?}");
    let last = numbers.pop();
    println!("{last:?}");

    let mut rows = vec![("pear", 2), ("apple", 1)];
    rows.sort_by_key(|row| row.1);
    println!("{rows:?}");

    let mut scores = HashMap::new();
    scores.insert("Ada".to_string(), 10);
    scores.insert("Grace".to_string(), 12);
    let ada = scores.get("Ada");
    let has_ada = scores.contains_key("Ada");
    println!("{ada:?} {has_ada}");
    for (name, score) in &scores {
        println!("{name} {score}");
    }
    let removed = scores.remove("Grace");
    println!("{removed:?}");

    let sentence = "the cat and the dog";
    let mut counts = BTreeMap::new();
    for word in sentence.split_whitespace() {
        *counts.entry(word.to_string()).or_insert(0) += 1;
    }
    for (word, count) in &counts {
        println!("{word} {count}");
    }
}
```

`empty` is `true`. After the pushes, the line is `3 true 10`, and `missing` is `None`. The loop prints `item 10`, `item 30`, `item 20`. After adding one and sorting, the vec is `[11, 21, 31]` and `pop` returns `Some(31)`. `sort_by_key` prints `[("apple", 1), ("pear", 2)]`. `get("Ada")` is `Some(10)`. The two names can print in either order. `remove` returns `Some(12)`. The `BTreeMap` prints `and`, `cat`, `dog`, then `the`.

> **Idiomatic Rust:** `Vec` for a list, `HashMap<String, V>` for string keys, `BTreeMap` when the printout has to be sorted. `.get` when the index or the key might be absent.

Side by side

	PHP	Rust
List	`$n[] = 10`	`Vec<T>`, `push`
Read	`$n[0]`, missing is `null`	`[0]` panics. `.get` is `Option`
String keys	`$scores["Ada"]`	`HashMap`, `insert` with `.to_string()`, `get("Ada")`
Count	`($counts[$w] ?? 0) + 1`	`*counts.entry(word.to_string()).o += 1`
Order	Insertion order	`HashMap` is unspecified. `BTreeMap` is sorted

Try it

Open `src/main.rs` in the Playground. The last four lines are the word counts in sorted order: `and 1`, `cat 1`, `dog 1`, `the 2`. Change the sentence so a word repeats, and run it again. That word's count goes up, and the lines stay alphabetical.

Key takeaways

- `Vec<T>` is the list: `push`, `pop`, `len`, `is_empty`, `contains`, `sort`, `sort_by_key`. One element type.
- `v[i]` panics when `i` is out of range. `v.get(i)` returns `Option`. `for n in &v` reads, and `for n in &mut v` writes.
- `HashMap<String, V>` is the associative array. `get` returns `Option`. Build keys with `.to_string()`. `entry().or_insert` / `or_default`, `retain`, and `extend` cover common map edits.
- `HashSet` is the unique-value set (`array_unique` / `array_flip` in PHP).
- `HashMap` order can change between runs. `BTreeMap` prints keys in sorted order.

What's next: Strings at a glance

Strings at a glance

In PHP

There is one string type. `.=` appends. `strlen` counts bytes. `mb_strlen` counts characters. `$text[0]` reads the first byte, which for `héllo` happens to be `h`.

text.php

```
 <?php

$text = "héllo";
$text .= " world";
echo $text, "\n";
echo strlen($text), " ", mb_strlen($text), "\n";
echo $text[0], "\n";

$name = "Ada";
echo "Hello, $name\n";
```

`strlen` prints `12`. `mb_strlen` prints `11`.

In Rust

`String` owns the text and can grow. `&str` is a borrowed view: a literal, or the type of a function parameter such as `name: &str`. `format!` builds a new `String`. `push_str` appends. Ownership and borrowing are chapter 3. This lesson only names the two types.

`.len()` counts bytes, like `strlen`. `.chars()` walks Unicode scalar values, like `mb_strlen`. `text[0]` does not compile. A string index is a range of bytes:

```
 error[E0277]: the type `str` cannot be indexed by `{integer}`
  --> src/main.rs:3:25
   |
3  |     println!("{}", text[0]);
   |                         ^ string indices are ranges of `usize`
   |
   = help: the trait `SliceIndex<str>` is not implemented for `{integer}`
   = note: you can use `.chars().nth()` or `.bytes().nth()`
```

`&text[0..2]` on `héllo` compiles, then panics, because those two bytes split `é`:

```
 thread 'main' panicked at src/main.rs:3:25:
byte index 2 is not a char boundary; it is inside 'é' (bytes 1..3) of `héllo`
```

`.chars().next()` is the first character, as `Option`.

src/main.rs

```
 fn greet(name: &str) -> String {
    format!("Hello, {name}")
}

fn main() {
    let literal: &str = "héllo";
    let mut owned = String::from(literal);
    owned.push_str(" world");
    println!("{owned}");

    let message = greet("Ada");
    println!("{message}");
    let bytes = owned.len();
    let chars = owned.chars().count();
    println!("{bytes} {chars}");

    match owned.chars().next() {
        Some(first) => println!("{first}"),
        None => println!("empty"),
    }

    let raw = "  héllo world  ";
    let trimmed = raw.trim();
    let parts: Vec<&str> = trimmed.split(' ').collect();
    let joined = parts.join("-");
    let has_world = trimmed.contains("world");
    let swapped = trimmed.replace("world", "Rust");
    let upper = trimmed.to_uppercase();
    let lower = trimmed.to_lowercase();
    let starts = trimmed.starts_with("héllo");
    let parsed: Result<i32, _> = "42".parse();
    let left = String::from("mug");
    let right = String::from(" bowl");
    let joined_plus = left + &right;
    println!("{trimmed}");
    println!("{parts:?}");
    println!("{joined}");
    println!("{has_world} {starts}");
    println!("{swapped}");
    println!("{upper}");
    println!("{lower}");
    println!("{parsed:?}");
    println!("{joined_plus}");
}
```

`owned` prints `héllo world`. `greet` prints `Hello, Ada`. The lengths are `12` and `11`. The first character is `h`. `collect` gathers the pieces into the `Vec` named in the type. `trim` leaves `héllo world`, `split` then `join` prints `héllo-world`, `replace` prints `héllo Rust`, `to_uppercase` prints `HÉLLO WORLD`, `to_lowercase` prints `héllo world`, `parse` prints `Ok(42)`, and `+` prints `mug bowl`. After `left + &right`, `left` has moved: you cannot use it again. Prefer `format!` or `push_str` when you still need both sides.

> **Watch for:** `&text[0..2]` is a byte range. On `héllo` it panics inside `é`. Walk characters with `.chars()`.

> **Watch for:** `intval("abc")` returns `0`. `"abc".parse::<i32>()` returns `Err`, so the failure can't be missed.

> **Idiomatic Rust:** Take `&str` when a function only needs to read text. Return `String` from `format!` when the caller should own the result. `.chars()` when you mean characters.

Side by side

	PHP	Rust
Owned text	`string`	`String`
A view of text	`substr` copies	`&str`
Build a line	`"Hello, $name"`	`format!("Hello, {name}")`
Bytes	`strlen`	`.len()`
Characters	`mb_strlen`	`.chars().count()`
First unit	`$text[0]` is a byte	`text[0]` does not compile. `.chars().next()`

	PHP	Rust
Trim	`trim`	`trim()`
Split	`explode`	`split` , then `collect` into `Vec<&str>`
Join	`implode`	`join`
Contains	`str_contains`	`contains`
Replace	`str_replace`	`replace`
Upper case	`strtoupper`	`to_uppercase`
Lower case	`strtolower`	`to_lowercase`
Concatenate	`.`	`s1 + &s2` (moves `s1`), or `format!` / `push_str`
Prefix	`str_starts_with`	`starts_with`
Integer	`intval`	`parse::<i32>()`

Try it

Open `src/main.rs` in the Playground. Confirm `12 11` for `héllo world`. Change `"héllo"` to `"hello"` and run it again: the lengths become `11 11` . Put `héllo` back, then try `println!("{}", &owned[0..2]);` and read the char-boundary panic inside `é` .

Key takeaways

- `String` owns growable text. `&str` is a borrowed view, including literals and `name: &str` .
- `format!` builds a `String` . `push_str` appends. `s1 + &s2` concatenates and moves `s1` .

- `.len()` is bytes. `.chars()` is characters. `text[0]` does not compile.
- A range such as `&text[0..2]` is bytes, and it panics when it splits a character.

What's next: match and patterns

match and patterns

In PHP

`match` is an expression and compares with `===`. A missing arm throws `UnhandledMatchError` when that line runs. `??` is the stand-in for a missing value.

A function that takes `?int $age` handles both cases, the same way the Rust function below handles `Some` and `None`.

age.php

```
 <?php

function show_age(string $name, ?int $age): void
{
    if ($age === null) {
        echo "$name has no age set\n";
        return;
    }

    echo "$name is $age\n";
}

show_age("Ada", 36);
show_age("Bea", null);

$years = null;
echo $years ?? 0, "\n";
```

This `match` has no arm for `3`:

label.php

```
 <?php

function label(int $n): string
{
    return match ($n) {
        1 => "one",
        2 => "two",
    };
}

echo label(3), "\n";
```

```
Fatal error: Uncaught UnhandledMatchError: Unhandled match case 3
```

In Rust

`match` is an expression. Every value of the type needs an arm, or a `_` arm. Literals, `|`, and ranges are patterns. `3..=5` includes both ends. `6..10` excludes `10`. Exclusive ranges in a `match` have been stable since 1.80. `n @ 10..=20` binds the value that matched. `(x, y)` takes a tuple apart.

`&'static str` means the text is baked into the binary for the whole program. Lifetimes, including why `'static` is special, are chapter 3. For now, treat it as "a string slice that lives forever".

src/main.rs

```
 fn label(n: i32) -> &'static str {
    match n {
        0 => "zero",
        1 | 2 => "small",
        3..=5 => "mid",
        6..10 => "high",
        _ => "other",
    }
}

fn describe(point: (i32, i32)) -> String {
    match point {
        (0, 0) => "origin".to_string(),
        (x, y) => format!("{x},{y}"),
    }
}

fn main() {
    let kind = label(4);
    let high = label(7);
    let other = label(12);
    println!("{kind} {high} {other}");

    let origin = describe((0, 0));
    let moved = describe((2, 3));
    println!("{origin} {moved}");

    let band = match 12 {
        n @ 10..=20 => format!("in band {n}"),
        other => format!("outside {other}"),
    };
    println!("{band}");
```

```
}
```

`label` prints `mid high other`. `6..10` is `6`, `7`, `8`, and `9`, so `10` would be `other`. The tuple prints `origin` and `2,3`. The binding prints `in band 12`.

Drop the `_` arm and the program does not compile. This is `E0004` for the arms above:

```
 error[E0004]: non-exhaustive patterns: `i32::MIN..=-1_i32` and
`10_i32..=i32::MAX` not covered
 --> src/main.rs:2:11
  |
2 |     match n {
  |           ^ patterns `i32::MIN..=-1_i32` and `10_i32..=i32::MAX` not
covered
  |
  = note: the matched value is of type `i32`
help: ensure that all possible cases are being handled by adding a match arm
with a wildcard pattern, a match arm with multiple or-patterns as shown, or
multiple match arms
```

A guard is `if` after the pattern. The arm runs only when the pattern matches and the condition is true.

`Option` is `Some(value)` or `None`. `match` handles both. `if let Some(years) = maybe` handles the `Some` arm and skips the rest. `let Some(years) = age else { return; }` is the shape the chapter 1 calculator uses: leave the function when the pattern does not match. `unwrap_or(0)` is PHP's `??`.

src/main.rs

```
 fn age_or_zero(age: Option<i32>) -> i32 {
    age.unwrap_or(0)
}

fn show_age(name: &str, age: Option<i32>) -> String {
    match age {
        Some(years) => format!("{name} is {years}"),
        None => format!("{name} has no age set"),
    }
}

fn main() {
    let named = show_age("Ada", Some(36));
    let missing = show_age("Bea", None);
    println!("{named}");
    println!("{missing}");
```

```
    let maybe = Some(36);
    if let Some(years) = maybe {
        println!("if let {years}");
    }

    let age = Some(36);
    let Some(years) = age else {
        return;
    };
    println!("else {years}");

    let fallback = age_or_zero(None);
    println!("{fallback}");

    let size = match 150 {
        n if n >= 100 => "large",
        _ => "small",
    };
    println!("{size}");
}
```

`show_age` prints `Ada is 36` and `Bea has no age set`. `if let` prints `if let 36`. `let ... else` prints `else 36`. `unwrap_or` prints `0`. The guard prints `large`.

> **Watch for:** A missing arm is a compile error in Rust. PHP throws `UnhandledMatchError` when that line runs. A guard comes after the pattern, and only then.

> **Idiomatic Rust:** `_` when the remaining values share one result. `match` on `Option` when both arms matter. `let ... else` when `None` should leave the function. `unwrap_or` for a default.

Side by side

	PHP	Rust
Expression	`match`	`match`
Missing arm	`UnhandledMatchError` when the line runs	`E0004` at compile time
Several values	`1, 2`	`1 | 2`

	PHP	Rust
Range	not a pattern	`3..=5` and `6..10`
Optional age	`?int` and `=== null`	`Option`, `Some` / `None`
Default	`??`	`unwrap_or`

Try it

Open the second `src/main.rs` (the Option program) in the Playground. It prints `large`. Change `150` to `40` and run it again. The guard does not match, so the line is `small`.

Key takeaways

- `match` is an expression. Arms can be literals, `|`, `..=`, `..` (stable since 1.80), a binding, or a tuple.
- A guard is `if` after the pattern. The arm runs only when the pattern and the condition both hold.
- A missing arm is `E0004` and the program does not compile. PHP throws `UnhandledMatchError` when the line runs.
- `match` on `Some` and `None`, then `if let`, then `let ... else`. `unwrap_or` is `??`.

What's next: Input classifier

Input classifier

In PHP

You would walk a list, skip the holes with `??` or a null check, and tally with an associative array. This exercise is the Rust version of that tally. No enums and no structs. Those are chapter 4.

classify.php

```
 <?php

$inputs = [12, null, 0, -3, 101, null];
$counts = [];

foreach ($inputs as $input) {
    if ($input === null) {
        $counts['missing'] = ($counts['missing'] ?? 0) + 1;
        continue;
    }

    $label = match (true) {
        $input >= 100 => 'large',
        $input >= 1 && $input < 100 => 'positive',
        $input === 0 => 'zero',
        default => 'negative',
    };
    $parity = $input % 2 === 0 ? 'even' : 'odd';
    $counts[$label] = ($counts[$label] ?? 0) + 1;
    $counts[$parity] = ($counts[$parity] ?? 0) + 1;
}

ksort($counts);
foreach ($counts as $label => $count) {
    echo "$label $count\n";
}
```

In Rust

The list is a `Vec<Option<i32>>`. Three functions are yours to fill in. `classify` uses a guard and ranges:

- `n if n >= 100` is `large`
- `1..100` is `positive`

- `0` is `zero`
- anything else is `negative`

`parity` is `even` or `odd`. `summarize` counts each label in a `BTreeMap`, including `missing` for `None`. `&[Option<i32>]` reads the list without taking it, and `&inputs` lends the `Vec`. `Some(&1)` in the tests is a lent count. Chapter 3 explains `&`. `101` is odd, so this list has two evens (`12` and `0`) and two odds (`-3` and `101`).

`cargo test` fails until the three functions do that. `cargo run` then prints the counts in sorted order. The button loads the starter. The solution is collapsed under it.

src/main.rs

```
 use std::collections::BTreeMap;

fn classify(n: i32) -> &'static str {
    todo!("classify {n}")
}

fn parity(n: i32) -> &'static str {
    todo!("parity {n}")
}

fn summarize(inputs: &[Option<i32>]) -> BTreeMap<&'static str, i32> {
    todo!("summarize {}", inputs.len())
}

fn main() {
    let inputs: Vec<Option<i32>> = vec![Some(12), None, Some(0), Some(-3),
Some(101), None];
    let counts = summarize(&inputs);
    for (label, count) in &counts {
        println!("{label} {count}");
    }
}

#[cfg(test)]
mod tests {
    use super::{classify, parity, summarize};

    #[test]
    fn classifies_sign_and_size() {
        assert_eq!(classify(12), "positive");
        assert_eq!(classify(0), "zero");
        assert_eq!(classify(-3), "negative");
        assert_eq!(classify(99), "positive");
        assert_eq!(classify(100), "large");
        assert_eq!(classify(101), "large");
    }
```

```
    #[test]
    fn classifies_parity() {
        assert_eq!(parity(12), "even");
        assert_eq!(parity(0), "even");
        assert_eq!(parity(-3), "odd");
    }

    #[test]
    fn counts_categories() {
        let inputs = vec![Some(12), None, Some(0), Some(-3), Some(101), None];
        let counts = summarize(&inputs);
        assert_eq!(counts.get("positive"), Some(&1));
        assert_eq!(counts.get("zero"), Some(&1));
        assert_eq!(counts.get("negative"), Some(&1));
        assert_eq!(counts.get("large"), Some(&1));
        assert_eq!(counts.get("missing"), Some(&2));
        assert_eq!(counts.get("even"), Some(&2));
        assert_eq!(counts.get("odd"), Some(&2));
    }
}
```

Solution

src/main.rs

```
 use std::collections::BTreeMap;

fn classify(n: i32) -> &'static str {
    match n {
        n if n >= 100 => "large",
        1..100 => "positive",
        0 => "zero",
        _ => "negative",
    }
}

fn parity(n: i32) -> &'static str {
    if n % 2 == 0 { "even" } else { "odd" }
}

fn summarize(inputs: &[Option<i32>]) -> BTreeMap<&'static str, i32> {
    let mut counts = BTreeMap::new();
    for input in inputs {
        match input {
            None => {
                *counts.entry("missing").or_insert(0) += 1;
            }
            Some(n) => {
                *counts.entry(classify(*n)).or_insert(0) += 1;
                *counts.entry(parity(*n)).or_insert(0) += 1;
            }
```

```
        }
    }
    counts
}

fn main() {
    let inputs: Vec<Option<i32>> = vec![Some(12), None, Some(0), Some(-3),
Some(101), None];
    let counts = summarize(&inputs);
    for (label, count) in &counts {
        println!("{label} {count}");
    }
}

#[cfg(test)]
mod tests {
    use super::{classify, parity, summarize};

    #[test]
    fn classifies_sign_and_size() {
        assert_eq!(classify(12), "positive");
        assert_eq!(classify(0), "zero");
        assert_eq!(classify(-3), "negative");
        assert_eq!(classify(99), "positive");
        assert_eq!(classify(100), "large");
        assert_eq!(classify(101), "large");
    }

    #[test]
    fn classifies_parity() {
        assert_eq!(parity(12), "even");
        assert_eq!(parity(0), "even");
        assert_eq!(parity(-3), "odd");
    }

    #[test]
    fn counts_categories() {
        let inputs = vec![Some(12), None, Some(0), Some(-3), Some(101), None];
        let counts = summarize(&inputs);
        assert_eq!(counts.get("positive"), Some(&1));
        assert_eq!(counts.get("zero"), Some(&1));
        assert_eq!(counts.get("negative"), Some(&1));
        assert_eq!(counts.get("large"), Some(&1));
        assert_eq!(counts.get("missing"), Some(&2));
        assert_eq!(counts.get("even"), Some(&2));
        assert_eq!(counts.get("odd"), Some(&2));
    }
}
```

The starter panics on `todo!` until you replace the bodies. The solution's `cargo run` prints the same seven lines the PHP tally prints after `ksort`, in sorted order:

```
 even 2
large 1
missing 2
negative 1
odd 2
positive 1
zero 1
```

> **Watch for:** `101` is odd. `0` is even. `1..100` stops at 99, so 100 falls through to the guard. Write `1..=100` and 100 becomes `positive`.

> **Idiomatic Rust:** `BTreeMap` so the summary is sorted. `match` on `Option`. A guard for the large band, and a range for the rest of the positives.

Side by side

	PHP	Rust
A hole in the list	`null`	`None`
A tally	an associative array	`BTreeMap`
Sorted keys	`ksort`	the map is already sorted

Try it

Open the starter `src/main.rs` in the Playground, or paste it locally and run `cargo test`. The three tests fail on `todo!`. Replace `classify`, `parity`, and `summarize`. `cargo test` passes, and `cargo run` prints the seven sorted lines above.

Key takeaways

- The list is `Vec<Option<i32>>`. The tally is a `BTreeMap`.
- `classify` uses a guard and `1..100`. `parity` is even or odd. `summarize` counts `missing` for `None`.
- `cargo test` is the check. The printout is sorted because the map is a `BTreeMap`.

What's next: What you learned

What you learned

In PHP

A variable can change type, a string is one type, and an array is both a list and a map that keeps insertion order. `match` throws `UnhandledMatchError` when a value has no arm and that line runs. `7 / 2` is `3.5`. `"0"` and `[]` are falsy. `??` fills in a missing value.

In Rust

`let` is immutable until you say `mut`. A shadow in an inner block ends when the block ends, and the outer name comes back. `const` is an item, not a `let`.

Integers default to `i32`, floats to `f64`. `7 / 2` is `3`. `as` truncates. `try_into` returns `Err` when the value does not fit. `bool` has no truthiness. `char` is one Unicode scalar.

A function's last expression, with no semicolon, is the return value. A semicolon makes that value `()`.

`if` needs a `bool` and both branches share a type. `(1..=5).rev()` counts down. `5..1` is empty. `break 'outer` leaves the labelled loop.

A tuple packs mixed types. `[T; N]` fixes the length, and a constant index past the end does not compile. A `Vec` grows. `v.get(i)` is `Option`. A `HashMap` does not keep insertion order. A `BTreeMap` prints keys in sorted order.

`String` owns text. `&str` is a borrowed view. On `héllo`, `.len()` is bytes and `.chars().count()` is characters.

`match` is exhaustive at compile time. A guard is `if` after the pattern. `let Some(x) = opt else { return; }` leaves on `None`. The input classifier puts those pieces on a `Vec<Option<i32>>`.

Key takeaways

- Names, numbers, functions, and control flow are the first half.

- Tuples, arrays, `Vec`, `HashMap`, and strings are the collections.
- `match`, `Option`, and the classifier are the last piece. The quiz is the button on this page.

Take the chapter quiz from the button on this page. It opens a separate page. Twelve questions, and you need 80% to pass. After you submit, each question shows a one-line explanation.

What's next: Why ownership